Implementation and analysis of custom instructions on RISC-V for Edge-AI applications

AJAY KUMAR M, University College Dublin, Ireland
VINEET KUMAR, University College Dublin, Ireland
DEEPU JOHN, University College Dublin, Ireland
SHREEJITH SHANKER, Trinity College Dublin, Ireland

The growing popularity of open-source processors, such as RISC-V, is reshaping the landscape of System on Chips (SoCs). RISC-V has gained widespread popularity due to its capacity for adaptable expansion of its Instruction Set Architecture (ISA), catering to diverse application needs. This study investigates an ISA extension focusing on memory load and store operations, which are crucial for many applications in embedded systems, including Deep Neural Networks (DNNs). Continuous memory interaction within DNNs leads to increased power consumption and latency. We explored a new instruction for doubling memory access, achieving a reduction of around **50%** in clock cycles and approximately **30%** in power consumption during memory load and store operations while incurring only a minimal area overhead of approximately **4%** and is validated on a modified RISC-V platform. This study also suggests additional ISA extensions, a work currently in progress, to facilitate support for the bfloat16 data type on RISC-V architecture to reduce quantisation losses in resource-constrained devices.

CCS Concepts: • **Computer systems organization** → **System on a chip**; • **Hardware** → *Hardware accelerators.*

Additional Key Words and Phrases: RISC-V, Deep Neural Networks, Memory, Accelerators, Edge-AI

ACM Reference Format:
Ajay Kumar M, Vineet Kumar, Deepu John, and Shreejith Shanker. 2024. Implementation and analysis of custom instructions on RISC-V for Edge-AI applications. In *14th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART '24), June 19–21, 2024, Porto, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3665283.3665342

1 INTRODUCTION

DNNs have demonstrated notable success in identifying and deciphering patterns across various applications, including image and video processing. Despite their effectiveness, DNNs face limitations in computational complexity, power consumption, and resource demands, making them less explored in contexts with low-power use cases, such as Internet of Things (IoT) sensors and extreme-edge devices. These applications often operate under stringent energy and resource constraints. The current body of literature reflects a growing focus on edge AI. Initially designed to analyse 2D data, DNNs were primarily employed in tasks such as image classification and segmentation. Training a DNN demands a substantial amount of data, and inherently, these networks are characterized by their vast scale, containing thousands & millions of parameters [1]

As the deployment of IoT sensors is rapidly expanding, with billions of interconnected devices, the imperative is clear: the computational processing must be decentralized to the edge. Some notable works towards bringing the DNN computation to edge are [4], [12], [2]. In the context of deploying computation to the edge, addressing challenges such

Manuscript submitted to ACM

as cloud connectivity, privacy concerns, and latency is paramount. However, this deployment introduces additional complexities, particularly regarding the availability of essential power and hardware resources. Numerous efforts have been aimed at enhancing performance by addressing the memory bottleneck, whether through constraining memory access, implementing on-chip memories, or employing memories closer to the accelerator. Yet, these strategies necessitate trade-offs, such as increased area due to on-chip buffer implementation or by reducing accelerator flexibility to optimize memory access patterns. In [9], the authors investigate the memory requirements of CNNs, analyzing both on-chip memory size and off-chip memory bandwidth to understand their impact on accelerator design, and propose heuristic design points to optimize for select dataflow scenarios. NNAMC is a dynamic random access memory (DRAM) memory controller tailored for neural network accelerators. It optimizes memory access performance by dynamically adjusting address mapping schemes based on memory access characteristics [11]. Researchers have also explored compute-in-memory approaches [7], [13] and have deliberated on the challenges associated with such techniques [6]. Novel microarchitectural designs for implementing multilayer DNNs in crossbar arrays of analog memories are introduced in [3], with improvements in circuit efficiency showcased through source follower-based readout, array segmentation, and transmit-by-duration techniques. A comprehensive review of state-of-the-art tools and techniques for efficient edge inference is provided in [8], with a focus on four research directions: novel DL architecture and algorithm design, optimization of existing DL methods, algorithm-hardware codesign development, and efficient accelerator design for DL deployment. However, to the best of our knowledge, all existing works primarily approach the problem from the accelerator's perspective. This study aims to address computational power and latency requirements by reframing the problem as a generic memory access limitation, rather than focusing solely on optimizing the accelerator for incoming data traffic.

Load and store operations are fundamental to any computation, including DNNs. As detailed in subsequent sections, we explore an extension to the baseline RISC-V architecture. This extension enables loading double words (and, by extension, 8 bytes and 4 half-words) from adjacent memory locations in a 32-bit RISC-V system. We validate our approach through a C example executed on a modified RISC-V platform within the Codasip framework [5]. We also briefly discuss a forthcoming aspect of our research, which involves incorporating support for the bfloat16 data type. However, our current focus remains on elucidating the memory access-related extension and presenting its preliminary findings, as this aspect of our work is still ongoing. The initial results demonstrate promising advantages associated with this ISA extension. To the best of our knowledge, no implementation of 64-bit load/store on 32-bit RISC-V has been done before.

2 OBJECTIVES

The objectives of this research are as follows:

(1) to demonstrate the advantages of running DNN models on RISC-V by reducing memory access through the proposed ISA extension for load and store operations.
(2) to address the compiler updates necessary to harness the potential of the custom ISA extension concerning memory accesses.
(3) to conduct performance comparisons between this extension and baseline RISC-V core.
(4) to introduce an ISA extension to support bfloat16 data type on RISC-V.

3 BACKGROUND WORK

This study uses a feature-reduced, 4-stage pipeline RISC-V core, provided by Codasip [5]. Support for On-Chip Debugging (OCD) was integrated into this core to facilitate JTAG access. Subsequently, it was interfaced with program and data memory and compiled in the Codasip framework to generate FPGA-compatible HDL files. These HDL models were synthesized for implementation on a Digilent Nexys-4 FPGA platform. Finally, the core underwent testing by executing a modified MNIST CNN C code featuring two convolution layers. This successful execution provided assurance of a streamlined workflow, enabling the initiation of ISA extension development.

4 LOAD STORE DOUBLE

RV32I is a load-store architecture with loads encoded in the I-type format, and stores in the S-type format [10]. The implementation of the load double and store double instructions begins by grouping the 32 existing RISC-V 32-bit general-purpose registers (GPR) into pairs. These pairs are denoted as GPR0_GPR1, GPR2_GPR3, ..., GPR30_GPR31, effectively providing us with sixteen 64-bit registers.

The Codasip framework facilitates the inclusion of compiler hints, enabling the seamless integration of extended instructions into the compilation process. This feature ensures that the SDK and other software components can be updated accordingly to support the extended instruction set. Such a capability underscores the advantages of software-hardware co-design.

A significant aspect of this research is the focus on compiler updates, which plays a crucial role. It is imperative not only to ensure accurate compiler updates but also to verify and adjust them to optimize user code to harness the benefits of the new ISA extension. For instance, the updated compiler must be capable of identifying and substituting memory accesses in a manner that ensures their consecutive placement, enabling the utilization of the load double and store double functionalities of the underlying hardware. Failure to incorporate these specific compiler modifications results in either the omission or partial utilization of the custom ISA for memory accesses. At present, our demonstration of ISA utilization is limited to manually coding consecutive accesses in C. We are engaged in identifying the necessary modifications to extend this optimization to generic user C code. The memory stage of the pipeline forwards the load addresses via the bus interface and retrieves corresponding data, which are then written back to the 32-bit register pairs. Following the implementation of 64-bit register aliasing, the subsequent step involves updating the pipeline stages. Specifically, the decoder undergoes modifications to accommodate the newly introduced opcodes associated with load double and store double operations. Upon encountering the load double opcode, the decoder activates both the ALU and load unit within the Execute stage while causing a pipeline stall, necessitating the computation of the subsequent memory load address. This address is determined by adding the base address specified in the rs1 field of the instruction with an offset of 0x4 (or 0x1, 0x2 for byte and half-word, respectively).

This process effectively conserves a few cycles that would have otherwise been expended on additional load instructions. Additionally, it optimizes program memory usage by consolidating multiple load instructions into one, thereby reducing the number of load operations from two to one (and correspondingly from 8 or 4 for byte or half-word loads, respectively). Likewise, in the case of store operations, data is retrieved from the register pairs and then deposited into the memory address indicated by the rs1 field of the S-type instruction. The initial value from the register pair is stored at the specified rs1 address, while the subsequent value is stored at an address incremented by 0x4 (or by 0x1 and 0x2 for byte and half-word data, respectively). The assembly format for the new custom instructions is shown in listing 1

Listing 1. Assembly format of the custom load double and store double instructions

```
; Load double assembly instruction format
ldd rd, imm12(rs1)
; Here, rd is a 64-bit register realised using a 32-bit register pair
; imm12 is the sign-extended immediate value in the I-type instruction format
; rs1 is the base address
; This instruction loads values into rd from memory addresses (rs1+imm12) & (rs1+imm12+0x4)

; Store double assembly instruction format
std rs2, imm12(rs1)
; Here, rs2 is a 64-bit register realised using a 32-bit register pair
; imm12 is the sign-extended immediate value in the S-type instruction format
; rs1 is the base address
; This instruction stores values from rs2 into memory addresses (rs1+imm12) & (rs1+imm12+0x4)
```

5 EXPERIMENTAL SETUP

Both the baseline and extended RISC-V subsystems undergo compilation using the Codasip toolchain, which automatically updates compiler support. LLVM serves as the compiler in this scenario. A simple C test is authored, which involves declaring a 2D array, reading its elements, and storing them in another array. This test aims to verify whether the compiler recognizes and utilizes the extended instruction when deemed necessary. Compilation is conducted with O3 optimization enabled. The selection of a 2D array is deliberate to simulate image-like behavior, aligning with the ultimate objective of benchmarking an image classification DNN.

The exercise is replicated on both the baseline and the extended RISC systems, with the baseline RISC system being slightly different from the RISC-V system, supporting a lesser number of instructions. Both systems, referred to as μRISC and μRISC-V, have lower instruction count support. The μRISC is a basic architecture intended primarily to include only the essential instructions required for generating a C compiler. The μRISC-V incorporates the entire RV32IM instruction set, excluding On-Chip-Debugger support.

6 EXPERIMENTAL RESULTS

The C test undergoes compilation, and the resulting executable is subjected to profiling within the Codasip framework. The assembly code highlighted in Figure 1 and 2 illustrate the substitution of regular load and store instructions with custom load and store instructions. The profiling outcomes are depicted in table 2 and 3, showcasing a 47.6% and 49.5% decrease in the number of clock cycles required for the identical test case, accompanied by a 50% reduction in the number of instructions executed. Additionally, it demonstrates approximately an 11% increase in area overhead in terms of required logical and sequential elements, although these are optimized during synthesis, as depicted in Table 1 for Xilinx XC7A100TCSG-2L FPGA. As anticipated, there is a slight area overhead of approximately 4% due to additional FPGA hardware needed for the custom instruction. However, this overhead is a negligible trade-off for achieving approximately 50% reduction in instruction count and clock cycles.

7 WORK IN PROGRESS

An important aspect to consider is that the adoption of the proposed extension necessitates compiler updates, which can be a nuanced process. We have encountered challenges wherein certain codes fail to recognize our instructions or only partially utilize them. Efforts are currently underway to address these issues, and we anticipate conducting

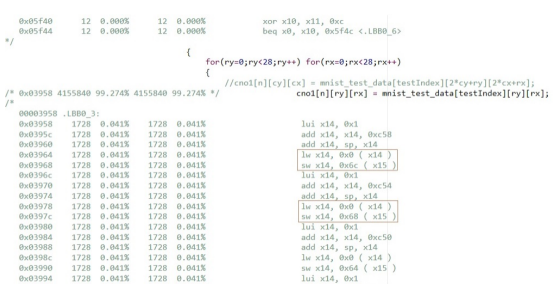

Fig. 1. Assembly code showing load and store instructions

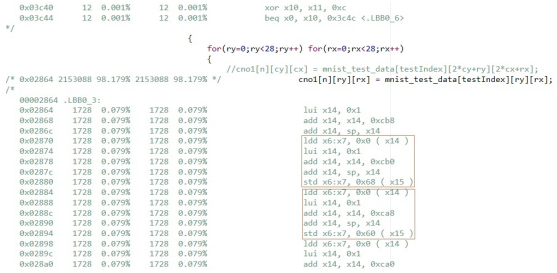

Fig. 2. Assembly code showing load double and store double instructions

Table 1. FPGA resource utilization comparison between baseline μRISC and extended architecture

	Baseline μRISC	μRISC with extension	Overhead (%)
LUT	1540	1605	**4.22**
MUX	309	213	**-31.06**
FF	1413	1414	**0.07**
DSP	3	3	**0.00**

Note: The synthesis results are only displayed for μRISC and not μRISC-V as that synthesis is still underway

Table 2. Performance metrics comparison between baseline μRISC-V and extended architecture

	Base μRISC-V	μRISC-V with extension	Change(%)
LIC	1.3597×10^6	0.6777×10^6	**50.15**
SIC	1.3599×10^6	0.6777×10^6	**50.16**
CC	4.1862×10^6	2.1930×10^6	**47.61**
HW	0.1194×10^6	0.1327×10^6	**-11.15**
TP	0.4206×10^{12}	0.2957×10^{12}	**29.70**
PC/C	0.1004×10^6	0.1348×10^6	**-34.26**

LIC = Load Instruction Count; SIC = Store Instruction Count; CC = Cycle Count; HW = Hardware area; TP = Total Power; PC/C = Power Consumption per Cycle

Table 3. Performance metrics comparison between baseline μRISC and extended architecture

	Base μRISC	μRISC with extension	Change(%)
LIC	1.3561×10^6	0.6777×10^6	**50.02**
SIC	1.3561×10^6	0.6777×10^6	**50.02**
CC	4.0750×10^6	2.0545×10^6	**49.58**
HW	0.0266×10^6	0.0296×10^6	**-11.27**
TP	0.1288×10^{12}	0.1126×10^{12}	**12.57**
PC/C	0.0316×10^6	0.0548×10^6	**-73.41**

LIC = Load Instruction Count; SIC = Store Instruction Count; CC = Cycle Count; HW = Hardware area; TP = Total Power; PC/C = Power Consumption per Cycle

application benchmarks in the near future to validate the efficacy of our approach. In our upcoming research, we aim to implement support for the bfloat16 data type on RISC-V architecture, following its introduction by Google. bfloat16 offers a promising blend of precision and storage efficiency, striking a balance between float16 and float32. This reduced precision is particularly advantageous for DNN training, as it introduces beneficial noise, thereby enhancing the robustness and reliability of trained models. Our investigation aims to capitalize on these benefits to advance AI computing on RISC-V-based edge devices.

We will evaluate two approaches to achieve this goal:

(1) Updating the compiler to include support for bfloat16 and subsequently designing a specialized hardware computation engine capable of processing this data type.
(2) Retaining the compiler without modifications and delegating the handling of regular data types to hardware, from conversion to computation. This will be facilitated through an ISA extension, offering users the flexibility to choose their preferred data type.

8 CONCLUSION

The custom instruction proposed in this research yields a notable reduction in both clock cycles and instruction count, presenting considerable advantages for executing AI workloads, particularly in edge computing scenarios characterized by strict power and resource constraints. While this work introduces the load double word extension, ongoing efforts are directed towards further extensions encompassing load double half word and load double bytes. Such extensions hold particular significance as many edge AI workloads employ quantized models that may not exclusively utilize 32-bit data. Additionally, there are intentions to expand the project to accommodate non-consecutive loads. This expansion will involve dividing the immediate field of the I-type RISC-V instruction into two segments, enabling distinct address offsets. Implementing this modification will necessitate updates to the compiler, the decoder and the execution unit. Moreover, the proposed extension is versatile and can be seamlessly integrated with other custom ISA extensions, such as the Multiply-Accumulate (MAC) operation, to unlock additional benefits. In summary, power consumption is a critical concern in embedded AI, and this study aims to address it by exploring a custom instruction for memory load and store operations in RISC-V, thereby reducing memory accesses and clock cycles. Additionally, we advocate for the incorporation of the bfloat16 data type, which is gaining traction for its ability to enhance model accuracy without significantly increasing resource requirements or computational complexity.

ACKNOWLEDGMENTS

This work was funded by Science Foundation Ireland through the SFI Centre for Research Training in Machine Learning (18/CRT/6183).

REFERENCES

[1] Mahbubul Alam, Manar D. Samad, Lasitha Vidyaratne, Alexander Glandon, and Khan M. Iftekharuddin. 2019. Survey on Deep Neural Networks in Speech and Vision Systems. arXiv:1908.07656 [cs.CV]

[2] Mohammadhossein Askarihemmat, Sean Wagner, Olexa Bilaniuk, Yassine Hariri, Yvon Savaria, and Jean-Pierre David. 2023. BARVINN: Arbitrary Precision DNN Accelerator Controlled by a RISC-V CPU. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASPDAC '23)*. ACM. https://doi.org/10.1145/3566097.3567872

[3] H.-Y. Chang, P. Narayanan, S. C. Lewis, N. C. P. Farinha, K. Hosokawa, C. Mackin, H. Tsai, S. Ambrogio, A. Chen, and G. W. Burr. 2019. AI hardware acceleration with analog memory: Microarchitectures for low energy at high speed. *IBM Journal of Research and Development* 63, 6 (2019), 8:1–8:14. https://doi.org/10.1147/JRD.2019.2934050

[4] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232

[5] Codasip. 2023. Codasip^{TM}. https://codasip.com/

[6] Chuan-Jia Jhang, Cheng-Xin Xue, Je-Min Hung, Fu-Chun Chang, and Meng-Fan Chang. 2021. Challenges and Trends of SRAM-Based Computing-In-Memory for AI Edge Devices. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 5 (2021), 1773–1786. https://doi.org/10.1109/TCSI.2021.3064189

[7] Anni Lu, Xiaochen Peng, Yandong Luo, Shanshi Huang, and Shimeng Yu. 2021. A Runtime Reconfigurable Design of Compute-in-Memory–Based Hardware Accelerator for Deep Learning Inference. *ACM Trans. Des. Autom. Electron. Syst.* 26, 6, Article 45 (jun 2021), 18 pages. https://doi.org/10.1145/3460436

[8] Md. Maruf Hossain Shuvo, Syed Kamrul Islam, Jianlin Cheng, and Bashir I. Morshed. 2023. Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review. *Proc. IEEE* 111, 1 (2023), 42–91. https://doi.org/10.1109/JPROC.2022.3226481

[9] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, and Andreas Moshovos. 2018. Memory Requirements for Convolutional Neural Network Hardware Accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 111–121. https://doi.org/10.1109/IISWC.2018.8573527

[10] RISC-V spec. 2017. The RISC-V Instruction Set Manual. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[11] Rongshan Wei, Chenjia Li, Chuandong Chen, Guangyu Sun, and Minghua He. 2021. Memory Access Optimization of a Neural Network Accelerator Based on Memory Controller. *Electronics* 10, 4 (2021). https://doi.org/10.3390/electronics10040438

[12] En-Yu Yang, Tianyu Jia, David Brooks, and Gu-Yeon Wei. 2021. FlexACC: A Programmable Accelerator with Application-Specific ISA for Flexible Deep Neural Network Inference. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 266–273. https://doi.org/10.1109/ASAP52443.2021.00046

[13] Shimeng Yu, Hongwu Jiang, Shanshi Huang, Xiaochen Peng, and Anni Lu. 2021. Compute-in-Memory Chips for Deep Learning: Recent Trends and Prospects. *IEEE Circuits and Systems Magazine* 21, 3 (2021), 31–56. https://doi.org/10.1109/MCAS.2021.3092533